UNIVERSITAT POLITÈCNICA DE CATALUNYA

MODELLING EFFECT OF NOISE IN DIMEMAS SIMULATION

Author

Chetan KC
chetan.kc@bsc.es

Supervisor

Jesus Labarta Mancho
jesus.labarta@bsc.es

A thesis submitted in partial fulfillment for the
Degree of Master In Research and Investigation
Specialization in Computer Networks and Distributed System

from the
Facultat d'Informática de Barcelona
Acomplised in
Tools Team of Department of Computer Science at BSC

April 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informática de Barcelona

Degree of Master In Research and Investigation

by Chetan KC

The applications that run on the HPC systems suffers the noise and all the applications has to coexist with this effect that has a different impact depending on the applications characteristics. This thesis approach is devoted to implement parametric statistical NOISE MODEL in Dimemas simulator in order to evaluate how the noise can distort the overall execution. Identify the major factors that are responsible for the performance degradation of applications and also develop the methodology to gather information of those factors in order to analyze the sensibility of applications towards noise.

Acknowledgements

At first I would like to be thankful to Prof. Jesús Labarta to bring me the opportunity to do this master thesis at BSC. Specially I want to mention Judit Gimenez, who have provided an important support for the accomplishment of this work. Finally I am thankful to every person from BSC tools team that did not hesitate to aid me during this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is dedicated to analyze the sensibility of application towards noise using Dimemas. The noise has been injected in computation and communication to see the effect of it in applications. This chapter contains motivation for the research, contribution and expectation from this research. It also contains how the rest of the document is organized.

1.1 Motivation

Super computing has been in Apex since couple of decades in any kind of scientific research. Today the fastest supercomputer[2] Sunway TaiHuLight System is of 93.01 Pflops with 40,960 nodes and 10,649,600 cores and total of 1.31 PB of memory, where the first supercomputer Cray-1 in 1975 [3] was of 160 MFLOPS with 1 million words of main memory. The figure 1.1 shows trend of supercomputing since 90's.

Parallel programming is also getting popularity[4] along with super computing, so that the applications could use the computation power of those HPC machines. All these enormous HPC machines uses linux operating system[2]. Parallel jobs are usually executed on such clusters which have linux OS by spawning one processor per CPU, and running to completion with no interference[5]. These parallel or HPC applications are often synchronous, which means every participating process is composed of computation phase separated by barriers. In synchronous system the fastest processes have to wait for slowest one to catch up. In this type of context the system began to experience difficulties in scaling applications to make use of hundreds to thousands of nodes. In [6][1] [7] have mentioned and analyzed how the system noise is affecting the super computing and parallel computing. OS noise might not noticed clearly in sequential applications

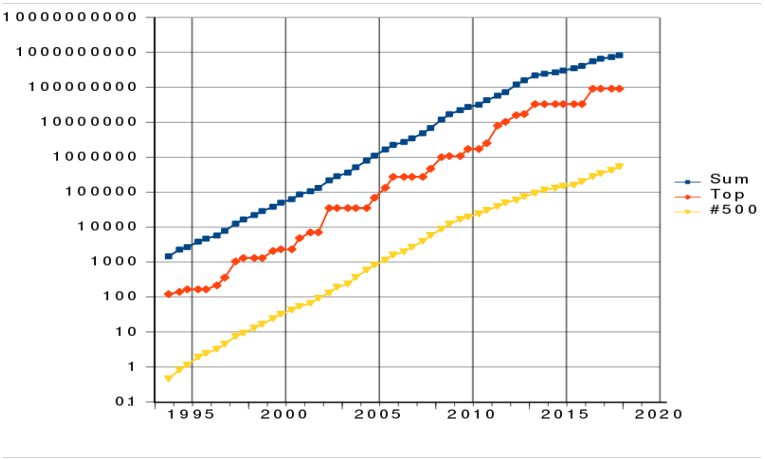

FIGURE 1.1: *Growth of supercomputer performance,Based on data from top500.org site. The logarithmic y-axis shows performance in GFLOPS*

because the processes are independent, but in case of synchronous jobs the affect can be noticed clearly because all the processes have to wait for the delayed one to catch up. This problem is from the beginning of HPC and now as the trend to the parallel programming is increasing, this noise is becoming a negative factor for the scalability of parallel applications.

Lightweight and micro kernels are common approaches taken to reduce OS noise on HPC system. It worked well when HPC system only required scientific libraries and a message passing interface (MPI) implementation. NAS is pretty famous benchmark suit and contains these kind of applications. Modern HPC applications are becoming more complex eg, UMT[1] from LLNL Sequoia benchmarks[8] and inclined towards memory sharing approaches, eg: OpenMP[9], Partitioned Global Address Space(PGAS)[10] such as UPC, Charm++, etc. Mean while there also exist another programming approaches like dynamic libraries, python scripts, dynamic memory allocation and virtualization seeing wider use in HPC systems. All these mentioned techniques require efficient support from the OS and system software. The trend had changed, nowadays the applications to run in the super computer are not coming only from the scientific domains, they are also from the new emerging fields such as financial, data analytics or recognition mining etc, to support all these we even need more powerful systems. The possible solutions might be to use the lightweight kernel to support all those mentioned applications, or a general purpose OS,(eg.Linux or Solaris) to the HPC domain, another solution might be to run the general purpose OS in virtual environment. It would be more relevant if we could be able to know the sensibility of application towards different kinds of machines which have different hardware and software configurations.

[1]MPI+OpenMP parallel scaling efficiency
threading compiler, single CPU performance and some python functionality

Dimemas[11] is a simulator for MPI application which can simulated applications. As discussed above the real machines has lots of OS noise, which was not modeled in Dimemas, so we came up with the idea to add a noise model in Dimemas, by which we can analyze the scalability of parallel applications in different configuration models. Which also gives us the approximation of performance in various system environment. OS noise might not be only the problems for scalability of applications in huge supercomputers, it might be affected by the communication between cores or nodes.

1.2 Noise Distribution

The authors in [1] have break down the noise in different categories. The figure 1.2 shows what could be the noise that affect any execution of application.

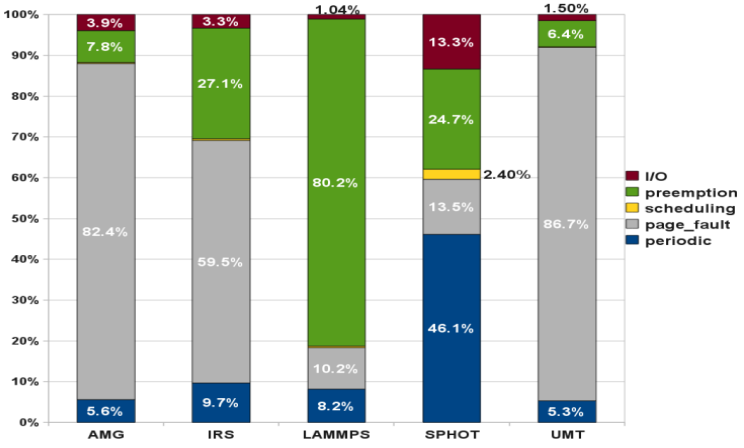

FIGURE 1.2: *noise breakdown for Sequoia benchmarks.[1]*

1. Periodic

   This noise could be caused by timer interrupt handler and Run_timer_softirq. For each occurence of timer interrupt triggers many activities like *Updates the time elapsed since system startup, Updates the time and date, Determines how long the current process has been running on the CPU and preempts it if it has exceeded the time allocated to it, Updates resource usage statistics, Checks whether the interval of time associated with each software timer.* Among which the most urgent is Update the time elapsed since system startup.

   Softirq is mechanism to handle processing that is almost as important as the handling of hardware interrupts. They run with high priority but with hardware interrupts enabled. They thus will normally preempt any work except the response to a "real" hardware interrupt.

2. Page Fault

   It occurs when a program try to access a block of memory that is not stored in physical memory. The fault notifies the OS that is has to fetch it from the storage, then transfer it from storage to the memory. An invalid page fault or page fault errors occurs when the operating system cannot find the data in virtual memory. This process is un noticeable in sequential programs, but the affect in parallel application in super machines can notice significantly. In Figure. 1.2 we can see page fault is one of the major source of noise.

3. Scheduling

   All the processes that need or want to be executed, must be in memory, and also need to have a CPU access. The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. It might take place when a process:

   (a) Switches from running to wait state

   (b) Switches from ready to running state

   (c) Switches from waiting to ready

   (d) Terminates

   According to above figure 1.2 scheduling don't seems the main source of noise in the OS, but in a really big system it might affect some how. There are several Scheduling algorithms such as : (a) First Come First Serve (b) Shortest Job First (c) Priority Based Scheduling (d) Round Robin

4. preemption

   The ability of OS to preempt (i.e: stop pause) a current scheduled task in favor of a higher priority task without requirement of co-operation, with the intention of resume it back in later time. This kind of change in the executed task are known as context switch.

5. I/O

   To make OS work properly, data paths must be provided that let information flow between CPU(s), RAM and score of I/O devices. The I/O of any system can be subdivided into user-space and kernel.

These are the major source of noise, which are almost impossible to remove completely. Where page fault and I/O are noise generated by applications which are out of scope of this research.

1.3 Tools Used to Analyze Performance

This thesis has been accomplished in **Performance tools** (Research and Development group) inside Computer Science Department of **BSC**(Barcelona Supercomputing Center). To achieve our goal we have used the tools (Dimemas, Extrae and Paraver) from this group along with NAS benchmarks[12].

1.3.1 Dimemas

Dimemas is a simulator for MPI applications developed by Tools team inside the department of Computer Science in BSC. Dimemas is Spanish name whose literal translation in English is "tell me more". The name Dimemas can be elaborate as DIstributed MEmory MAchine Simulator. It is a simulator with which we can simulate the applications with taking in consideration the key factors such as latency, bandwidth, contention etc.. The fundamentals parameters are as follows:

(i) Machine : number of machines you want in your workstation.

(ii) Node : number of nodes per machine.

(iii) Core : number of cores(processors) per node.

(iv) Latency : From the network or may be because of the overhead of any process before arrive to the Network.

(v) Bandwidth : This can be sub divided into two parts (a) Inter node bandwidth is the bandwidth used by nodes to communicate between them, which we also called the network bandwidth and (b) Intra node bandwidth is the bandwidth inside node, which is called as memory bandwidth. This bandwidth is used by cores to communicate among them inside one node.

(vi) Contention : How many messages are allowed in a network at a time. The contention might be in two different place one in a inter node communication and another one is on intra node.

To be able to simulate, dimemas needs the trace file of application in its own format. We have "prv2dim" to convert the trace files into the required format for simulation. Once we have a trace file of application converted into a required format, we need another file defining the configuration environment in which we want to simulate our application. In configuration file we have to defined all the parameters mentioned above. The main

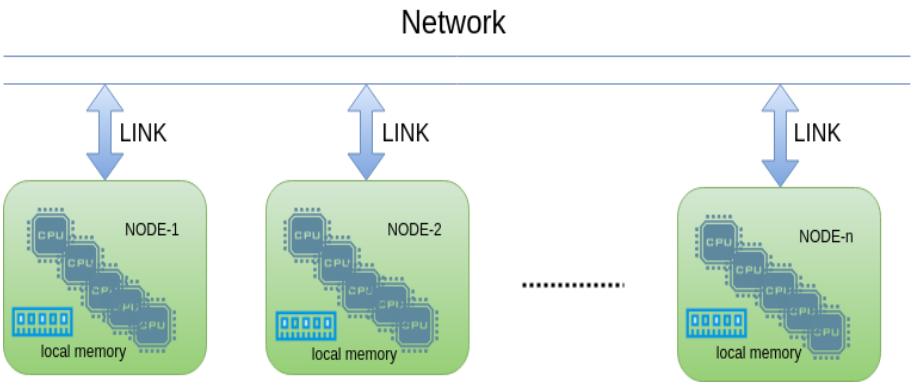

FIGURE 1.3: *Structure of Dimemas*

purpose of the Dimemas simulator is to figure out the question **what if in case?** but still there are some aspects that we can improve.

1.3.2 Extrae

The name Extrae in spanish word means "to extract", in this case to extract a trace file of applications. It is a tool which uses different interposition mechanism to inject inquest into the desired application in order to gather information regarding to its performance. We have used LD_PRELOAD mechanism to generate the trace files of the application that we wanted to analyze. It is one of the interposition mechanism used by extrae to generate the trace file of application. It support many programming models (i) MPI (ii) OpenMP (iii) CUDA (iv) OmpSs (v) Java (vi) and Python It is also available for many platforms such as Linux, Android, ARM etc. for more information you can see in the website[2]. Extrae saves the gathered info a format(.prv) which can view from the visualizer tools called Paraver.

1.3.3 Paraver

The information collected by the extrae from the applications are stored in three different files. Instead of read all these files one by one we can use another tool "paraver", which express all those information of three files by means of visualization. The meaning of paraver in english is "to see" as its function. Among three files used by paraver, the first one one with extension .prv is a ASCII file which contains the list of all the records (states,events and communications). The second one is Paraver Configuration file which ends with extention .pcf which defines the labels and colors associated to states and events. The third file ends with .row which is Names Configuration File, which contains

[2]https://tools.bsc.es

the rows label or rows for the time line. We have used paraver to see the structure of applications, analyze them before and after simulation and also for validation.

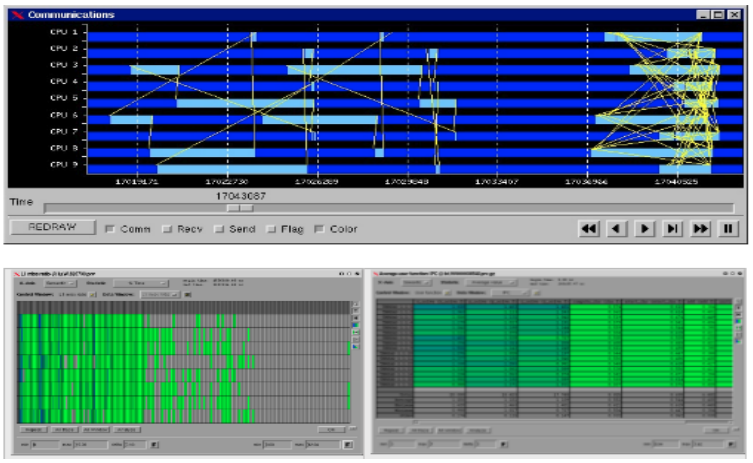

FIGURE 1.4: *Views of paraver*
Source: tools.bsc.es/paraver

Another feature of paraver is you can cut the trace files according to your interest or requirements or you can take a portion of trace file where you are interested in, the name of this feature is called cutter. We have also used this feature to cut the unnecessary part of trace files for our analysis.

The another feature that we have used of paraver is "paramedir" which meaning in english is "to measure". Paramedir is a non-GUI version of the Paraver, which use the same files as paraver but instead of express information by visual it saves it in ASCII file. The extension of the ASCII file can be defined by user. It uses the same configuration files that uses the paraver to visualize. In other words we can filter only the certain information that we are interested in. With the help of this feature we have collected information of application that we are interested in.

1.4 Contribution

In this thesis we added a new model called as **"noise model"** in Dimemas Simulator. Noise model is adding some sort of noise in during the simulation to see its it's effect on the applications. In this model we have injected noise mainly two different events 1. Communication and 2. Computation To inject a noise a random vaule is generated with normal random distribution. In case of communication we can subdivide it in three different steps (i) inter-node communication (ii) intra-node communication and (iii) in both communication.

In case of communication the obtained random values will be rested with the original bandwidth. If we are making penalties in inter-node communication, for each inter-node communication the random value will be generated and rested to the original bandwidth and the communication will be performed. The same process will be repeated while making the penalties in intra node communication. In case of making penalties in both communication in any communication the original bandwidth will be rested with the generated random values.

To inject a noise in computation the obtained random values will be added to the computation time for every cpu burst. To do so we first obtained the cpu_time for any job. Here cpu_time means we refer, how long the job is inside the cpu. As we get the cpu_time, the random values is added to that time. We have explained in detail how exactly we have make penalties in communications and injected noise in computation in methodology part.

1.5 Expectation

We are expecting with this model we would be able to know, the sensibility of the application towards noise. The impact of different configuration environment on the applications. The impact of different factors in the execution of parallel applications. Which factor(memory/network bandwidth or computation time) have more impact on application, from which the user could be able to modify some aspects which will improve its efficiency during the execution. And also discover the characteristics of the applications and able to choose the best configuration environment to execute those kind of applications.

1.6 Dissertation Organization

The rest of the document is organized as follows. Chapter two contains the previous work done in the similar field, chapter 3 explains how we have implemented our "noise model" in dimemas simulator, all the configurations that have been used during the experiments,the benchmark used and also the how we have executed our experiments. Chapter 4 has all the results that we obtained from our experiments and at the end chapter 5 contains the conclusion of this thesis.

Chapter 2

State of the Art

As a cluster get popularity in HPC(High Performance Computing) which size may vary from tens to thousands of nodes, which normally have general purpose operating system installed on them, almost all of them have Linux Operating System (OS). To take advantages of those tremendous machines, trend of parallel computation and parallel programming also get popularity. Parallel computing refers to the processing of multiple jobs simultaneously on multiple processors. The proper studied and analysis of parallelization of tasks and application and the system configuration might be helpful to decrease those time consuming activities inside the HPC machines. During all these years many studied have been done to analyze the performance of those HPC's, among them few are studied and resumed in this chapter.

2.1 Literature Review

Operating System (OS) noise (or jitter)is well known problem from the beginning of High Performance Computing(HPC). As the requirement of HPC is increasing exponentially the larger systems are building, the OS noise is a bottleneck for those large systems. By using the Fixed Time Quantum(FTQ)[1][18] benchmark and Linux Trace Toolkit Next Generation(LTTng)[19] did the quantitative analysis of OS noise in parallel applications [1] . The breakdown of noise in several small divisions and analyzed them separately is the most powerful part of this research. After analyzing the os noise with two different way FTQ and LTTng they have concluded that micro benchmarks are not able to distinguish two unrelated events if they happen in the same iteration, because they compute the OS noise as a missing operation in a given iteration, so overcome this

[1]Benchmark which measures the work per unit time in basic operations.

problem they have used LTTng . Page faults may even have larger impact than timer interrupts (both in terms of frequency and duration), the affect of jitter almost depends on the type of applications and some activities have larger time distribution which may lead to load imbalance at scale for some applications, which is the most important conclusion of this research.

The overall performance of any parallel application is consequence of both user-level execution code, OS level operation occurring during it execution and certainly in the interactions between them. If we would be able to study and measure all these three parameters, we could be able to know the performance of HPC machines and the scalability of application. Many different approaches have been tried and executed to measure the performance of parallel applications in HPC, among which Aroon and his co-workers[21] have extended the approach of KTAU in [22] kernel level measurement system which is able to capture and create a metrics from kernel performance data that can be read by the TAU[2][23] application level performance tool. Using the technique of KTAU they have analyzed the popular applications of NAS [12] LU and CG applications in IBM BG/L machine with injecting the noise by 1%,5%,10 % and with no noise. they have injected a nosie at a frequency of 1000HZ with every noise event costing 10 microseconds, which means total of 1%, in the case of 1% noise and 50 and 100 microseconds respectively for 5 and 10 % of noise. With these experiments they concluded the scheduler as a major source of noise with 85% of total noise. I missed a more source of noise are neglected as mentioned in [1] if it could be possible to study all of them, we could be more sure about the performance HPC systems over applications.

"Dan Tsafri" and his co-workers in 2005 [16] Explains how OS ticks clocks produce noise in the system and what is the impact of it in the parallel jobs. **System Noise** is becoming a huge problem as a parallel jobs are getting very large and finer in granularity. Mainly there are two types of activities which are responsible for the system noise **network** and **periodic clock ticks**. At a boot time the general-purpose kernel sets a hardware clock to generate periodic interrupts every few milliseconds which is called a tick. The experiment was based on the simple theoretical explanation for the well known empirical fact that noise effects increase with the cluster size. Let P be the per-node delayed probability for a running task by noise and n be the cluster size, if P is small enough then the noise is linearly proportional to n such that $P \times n$ is good approximation of the parallel job's probability to be delayed upon each computation phase. The P value seems very sensitive even in the minor change in hardware configuration. The direct overhead of ticks is relatively small (less than 1%),but the indirect overhead might reach more than tens of percentage depending on the size and type of cluster. As a alternative

[2]Portable profiling and tracing toolkit for performance analysis of parallel programs in C, C++, UPC, JAVA, Python.

for the ticks they suggest use of "smart timers" which allow accurate timing with a settable bound on maximal latency and reduce overhead by aggregating nearby events and by avoiding unnecessary periodic ticks.

As mentioned above super computing get popularity since 90's. In 2003 Petrini.. [6] did a research in newly installed supercomputer ASCI Q which were installed at Los Alamos National Laboratory(LANL), which have capacity of 20Tflops/s listed in number second most powerful supercomputer in list of top500 computers at that time, to figure it out either the machine would be able to run the applications as it should run. For their research they have use SAGE[3]. After executing the application they found that the performance in ASCI Q was quite poor, while using the all four cores per node, if they use only 3 cores per node the performance is quite good. The first conclusion of the research is leaving one processor per node free we can achieve better performance at least in ASCI Q. Poor performance on SAGE was not the fault of neither the MPI implementation or the network, there was some problem on nodes. Another important conclusion of this paper is "one fine-grained applications, more performance is lost to short but frequent noise on all nodes than to long but less frequent noise on just a few nodes ". They removed some noise caused by the daemons in system with this there were able to double the SAGE's performance. With this we can conclude that the OS noise could half the scalability of parallel applications. Not only the Petrini and his co-workers have concluded this type of conclusion kramer and Ryan[20] also observed the same kind of behavior with Embarrassingly parallel(EP)[12] benchmark running on the Pittsburgh Supercomputing Center's AlphaServer cluster. Because EP performs very little communication and should be robust to both processor mapping and network performance, the authors conclude node is the source of performance variability.

AMG [26] was introduced in 1980s, needing to solve the large problems posed on unstructured grids. It is composed of simulation in various areas such as *ground water flow, explosive materials modeling, electromagnetic applications,fusion energy simulation*, with "optimal" property. By taking the advantage of this Gadvari along with his team [27], tried to run the AMG algorithm in the HPC system to find out the performance of parallelization in HPCs and locate the bottleneck of it. They began with very simple model $\alpha - \beta$ for communication along with an alalytical model of the computation. They also added some more machine constrains including distance effects, reduced per core bandwidth, and number of cores per node. They executed AMG in different systems like Intrepid[4], Jagur[5], Hera and Zeus[6] and Atlas. After running several times

[3]A comprehensive Eulerian hydrodynamics code consisting of Fortran + MPI code
[4] IBM BlueGene/P system at Argonne National Laboratory
[5]hybrid Cray system at Oak Ridge National Laboratory
[6]Linux cluster at Lawrence Livermore National Laboratory

with all these architecture they come up with the conclusion that both the contention and the distance of communication are performance bottleneck for AMG. They have studied how the contention and the communication effect the parallel application in HPC but there also exist OS noise from the beginning which is also the major issue for performance bottleneck.

Normally every application in their initial phase is initialization and some communication to assign variables and few more simple actions so, this part of the application could be skipped and don't have any importance while analyzing the parallel efficiency. Computation phase of any application is the core phase where we can found the typical behavior promoted by the application's structure, which performs iterations on space and time. This is the phase where application shows its parallel efficiency. Therefore it is interesting to study this phase to know how the application is behaving towards the parallelization and load balance. In [7] using the signal processing techniques the MPI applications have been analyzed to see their parallel efficiency. The Discrete Wavelet transform (DWT)[7] is applied to the signal obtained from the several trace files(such as: Weather Research and Forecast(WRF), Non-hydrostatic mesoscale Model(NMM), Applied Research Weather(ARW)) and the execution was performed in MareNostrum supercomputer. They are only measuring the parallel efficiency of the application which is also our interest, and the cutting the core part of the application where we can find the related information for our research is the most interesting and related part for our work.

In 2014 Claudia Rosas and his teams at BSC[8] in [28] also have studied to identify the fundamental factors that are affecting in the parallel efficiency of the MPI application. Although they are not analyzing the effect of noise in the application, the parameters they are analyzing are the similar one that we are interested in. They also claimed that the communication doesn't affect as computation in parallelization and the variation in scalability of application was due to the system noise. Beside these papers in [15] they explained how the cpu power unbalance could produce the noise in during parallel execution. Another research which was done in 2012 [25] which explains the I/O could be the bottleneck for the huge size applications in HPC's. Both papers are interesting to know, how the parallel applications could be affected in execution, but remains outside the scope of this project

[7] Mechanism to filter only the core part of the applications to analyze their scalability
[8] Barcelona Supercomputing Center

Chapter 3

Methodology

We already mentioned in previous chapter that we want to analyze the effect of noise in applications using our simulator tool “Dimemas”. We begin the simulation from making penalties in communication, to see the reaction of applications towards bandwidth. In second part of the analysis we injected a noise in the computation using a same random distribution, the noise to be injected depends on the time taken by the processor to compute a certain task(time of computation). In this chapter you can found the explanation about module that we have added in dimemas along with its working mechanism. The application that were used to finalize this work are also explained in this part of the document, and also the description of the parameters that we wanted to analyze after executing. Finally the description of the system we have developed to run the simulation and save those parameters automatically.

3.1 System Configuration

It is more meaningful to know about the application(structure and characteristics) and also the important parameters of the system which will have impact on it. To finalize our goal ”effect of noise in the Dimemas simulation ” we have used a many different configuration along with NAS[12] benchmark.

3.1.1 Dimemas Configuration

For simulation of Dimemas we need two files 1. Application trace file converted into dimemas format prv2dim is used to convert it and 2. a system description file which kind of system environment we want to run our simulation. In second file we can configure

almost every important parameters of the system which have impact on the execution of application. The most relevant and important parameters for our simulation can be divided into three different section, "environment configuration", "node configuration", and "mapping configuration". Following tables shows the values, parameters and the description of each of them.

Parameter	Definition	Value
Number of nodes	number of nodes in one machine	1024
Intra node bandwidth	bandwidth to communicate between nodes	1024.0 MB/s
Inter node buses	0 means no contention and N means "N" number of bus contention	0

TABLE 3.1: *Environment configuration for dimemas*

In this section we configure the general environment, where we defined the number of nodes we want to have in our system and communication bandwidth between them. The last parameter in this section is number of buses we allow to have in order to communicate between nodes, for our simulation we put it "0".

Parameter	Definition	Value
Number of cores	Number of processors per node	64
Intra node latency	communication latency within node (between cores)	4μs
Intra node bandwidth	communication bandwidth between cores inside a node	1024 MB/s
Intra node buses	max number of message at a time inside a node	0 (unlimited)
Intra node input links	number of intra node input links	1
Intra node output links	number of intra node output links	1
Inter node latency	communication latency within nodes	4μs
Inter node bandwidth	communication bandwidth between nodes	1024 MB/s
Inter node input links	number of inter node input links	1
Inter node output links	number of inter node output links	1

TABLE 3.2: *Node configuration for dimemas*

The TABLE 3.2 demonstrate the configuration of a node that we have used for our experiments. Where we used 4 μs [1] of communication latency in both (inter and intra) node communication. Contention (number of messages could be in a link) is set to 1

[1] micro seconds

both for inter and intra node communication. The value "0" means without contention and "1" means with one contention. Network and memory both bandwidth set to be 1024 MB/s. These are the main factors inside the node which have direct effect on the execution.

Parameter	Definition	Value
Trace file name	Name of the trace file (optional)	"file_name"
Map Identifier	How you want to fill up the nodes and how many tasks per node	N tasks per node

TABLE 3.3: *mapping configuration for dimemas*

After configuring the system we need to mention how we want to be mapped the tasks in the system. Table 3.3 shows two different parameters where we can choose accordingly. Trace file name is optional, where Map identifier is compulsory. Map identifier defines how we gonna distributed tasks in the machines during the simulation. It has 3 different options to assign the tasks to the nodes. All of them have direct meaning: 1. Fill nodes: it fills completely the first node with equal number of task as the number of processors in that node and it goes in ascending order till the number of nodes. 2. N taks per node: will fill the nodes according to the value of N. we used this option to execute our experiments. Because it allow us to put different number of tasks per node without changing any other configuration. 3. Interleaved, will fill one node and leave the second one and map to the third one and it goes like this till the end.

For our experiments we used "N tasks per node" as node identifier as shows in table 3.3 (where N = 1,4,8,16,32,64) tasks per node with above configuration for all benchmarks. Which means we run one application with 6 different configuration.

3.1.2 Random Configuration File

This file is source for a random value generation, which we will inject as a noise. In this file we mention where and how much noise we gonna inject and also a distribution to generate the value. In Table 3.4 you can see all the parameters, it's description their values that were used in the experiment.

The main parameters of the random file are Network and memory bandwidth and the computation, where we are injecting the noise. A normal random distribution is used to generate the random values with fixed mean and varying the standard deviation. In case of communication we have tried with lower values then 300 MB/S as a mean but the applications that we have used here almost do not shown any affect. We pick up

Parameter	Description	Distribution name
Computation	This field represent a noise in computation	normal int (30000 nanoseconds,var)
Network_bandwidth	this field set means penalties in inter-node communication	normal int (300 MB/s,var)
Memory_bandwidth	this field set means penalties in intra-node communication	normal int (300 MB/s,var)

TABLE 3.4: *Random file configuration for simulation*

300 MB/s as a mean because from this point they started to show the effect of it in the execution. The standard deviation "var" will be vary for every random file generation starting from "75 MB/s" till "950 MB/s" with constant number of "75 MB/s". From which we will have 12 different random files with different standard deviation and fixed mean of 300 MB/s. We have used 75 as a starter and and also to increment the standard deviation because 75 is neither too big nor too small to see the effect of it injected noise in application. We have tested with smaller values and bigger ones but with them either the application need lots of simulation to show the effect or the affect is too big from one simulation to another. As we have setup our both communication bandwidth as 1024 MB/s, we believe 950 MB/s is big enough to see any effect due to communication. **Note** all the values that will be generated with random files are in MB/s for communication.

With same reason as in communication we have used fixed mean as "30000" nanoseconds and varying the standard deviation starting from "1000" till "1000000" nanoseconds incrementing with a constant of "10000" nanoseconds. With these values we get 10 different random files for computation.

3.2 Noise Model

Our main objective is to add a noise model in dimemas to see the effect of it on parallel execution of applications. To explain this model we can subdivide this model in two different parts. The first part is to make a penalties on communication and the second one is to inject a noise in the computation. Algorithm 1 shows how we check either we are injecting the noise or not, if yes in which part.

Algorithm 1 is responsible to check either we are adding noise in the simulation or not. As in algorithm if random flag(r) is not mentioned or there is no value in the random

Algorithm 1 Noise Simulation

```
1: procedure START SIMULATION
2:     Check the random flag r
3:     if flag(r) == 0 then
4:         go to Contd.. simulation
5:     else
6:         read the random file
7:         check which parameter is/are set
8:         read the values and call the random generator
9:         go to Contd.. simulation
10:    Contd.. simulation
11:    End
```

file then the simulation is done without adding anything, in another words, with the original configuration. If there is random flag(r) set in simulation command then it will check the parameters in the random file and call the random generator function. Here it also check in where we are injecting the noise, in communication or in computation or in both.

3.2.1 Noise In Communication

After confirming that we are injecting a noise in communication from random file parameters, this algorithm will be called. The noise in communication can be subdivided into two different parts: 1. inter-node and 2. intra node communication. We have used a simple way to make a penalties in the communication. Algorithm 2 shows how the penalties are made in every inter-node communication.

Algorithm 2 Noise in inter-node Communication

```
1: if random file parameters == network then
2:     for every_inter_node_communication do
3:         noise(n) MB/s ← calculate value with given parameters
4:         if noise(n) <= 0 OR noise(n) >= original_bandwidth then
5:             new_bandwidth ← original_bandwidth
6:         else
7:             new_bandwidth ← original_bandwidth-noise(n)
8:         Do communication with new_bandwidth
9: End fxn
```

The algorithm 2 shows how the we have implemented to make penalties in the inter-node communication which is also called network communication. If the parameters are set for the inter node communication, the generated random value is assigned as a noise. After assigned to the noise it check either it is less then zero or greater than the original bandwidth. If any of the condition matched then we set the new_bandwidth to the original. If non of the above condition is matched then we rest the generated value in the original bandwidth and set it as a new_bandwidth, and the communication will be performed with the new_bandwidth. The same process will be repeated for every inter-node communication.

In the case of intra-node communication the process is exactly same as in the inter-node, it will be active if in random configuration file the memory communication parameters are set and of course the random flag in simulation command.

We have also made penalties on both communications(intra and inter node), just by setting the values for both parameters and by setting the random flag on in simulation command. If both the parameters are set then it will call the function accordingly depending on the communication the processor is doing. The application suffers much more if we made penalties in both the communications, detailed results are explained in next chapter.

3.2.2 Noise In Computation

The another thing that we wanted to analyze the sensibility of application towards noise in computation. Algorithm 3 explains in detail how we have implemented our system to inject a noise in the computation.

This algorithm 3 is called when the computation parameters are set in random file. As the function is called at first it generates the random value with the values in random file. Once the value is generated it will check either the generated value is smaller or equal to zero, if it is smaller or equal to zero then there will be no noise, otherwise the generated value will be added in the execution time which we called the noise. The unit for generated value is nanoseconds. Here we have injected the noise in the execution time depending on how long the processor takes to execute the certain burst or how long the burst last on a CPU. We also defined the ideal time ie: "1 milliseconds" which means if any burst remains 1 millisecond or more on CPU, then by sure it gonna have some random noise in its computation time other wise we calculate the probability either it gonna have a noise or not.

Algorithm 3 Noise in computation

1: **if** random files parameters == computation **then**
2: **for** *every_compuataion* **do**
3: noise(n) (*in nanoseconds*) ← generate value with given parameters in random file
4: **if** noise(n) <= 0 **then**
5: *noise(n) == 0*
6: **Get** execution_time
7: **if** *execution_time*< = PHY **then** ▷ PHY=100000 nanoseconds
8: calculate the probability(P) = $\frac{exexution\_time}{PHY}$
9: **else**
10: probability(P) ← 1
11: **set** inject_noise as boolean
12: *check probability either it is between "0" and "1" and set inject_noise accordingly*
13: **if** 1 = <*inject_noise*> = 0 **then**
14: execution_time ← execution_time + noise(n)
15: **else**
16: execution_time ← execution_time
17: Return execution_time
18: **End fxn**

The function get a execution time of a burst and check the time how long it remains inside a CPU. If execution time is smaller then 1 millisecond or 100000 nanoseconds then, it calculate the probability to inject a noise in that computation otherwise the probability will be 1. The probability is calculated as shown in algorithm 3, which will be utilized to inject the noise in computation. Depending on the probability if it is in-between "0" and "1" then we add a noise(n) in the execution time otherwise execution time will remain as it was. Dimemas calculate everything in nanoseconds so the unit of time we are working during the simulation will be in nanoseconds.

3.3 Paramedir Configuration

In introduction part we already mentioned that paramedir is non_GUI version of the paraver. Which uses the same configuration files that uses the paraver, but paramedir saves all those information in a ASCII file instead of showing them in different views. As paramedir also use the same configuration files that the paraver uses we used some of the existing configuration file to collect the interested data's. We also configured

some configuration files by ourselves with the help of paraver to collect the rest of the information from the application that we are interested in.

To make a configuration file for paramedir is done as follows: Firstly open the trace file with the paraver, change the views so that you can see the parameters that you are interested in, as the paraver view window shows the parameters what you are looking for, right click on the view window and save it as a configuration file. Once you have a configuration file you can use it through paramedir to collect the same parameters that you saw in the paraver views in a ASCII file.

3.4 Benchmarks

To achieve our objective we selected the 3 applications from the NAS[2] Parallel Benchmark(NPB) 3[12]. It is designed and developed to evaluate the performance of parallel supercomputers with implementation of MPI and OpenMP. It consist of five parallel kernels and three different simulated application benchmarks. All of them are designed with the same characteristics like computation and data movement of computational fluid dynamics(CFD) applications. We can generate the application in different sizes, here we have used class A which is standard test size. The brief description of three application that we have used for our experiments are as follows:

1. MG(Multi Grid)

   This is a simplified multi grid calculation, which requires highly structured long distance communication and tests both short and long distance data communication.

2. CG(Conjugate Gradient)

   In this kernel, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue. A kernel is typical of unstructured grid computations, which test irregular long distance communication, employing unstructured matrix vector multiplication.

3. BT(block Tri-digonal Solver)

   This benchmark performs a synthetic CFD problem by solving multiple, independent system of non diagonal dominant, block tridiagonal equations.

[2]Numerical Aerodynamic Simulation

CG and MG are the parallel kernels where BT is one of the pseudo application. We choose three different application of class "A" which have different characteristics to give a variation.

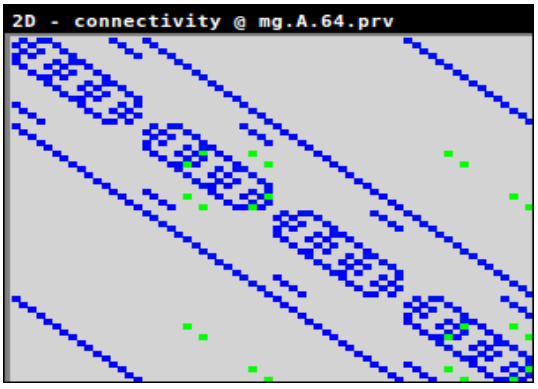

FIGURE 3.1: *Paraver view of p2p communication of mg benchmark with 64 tasks*

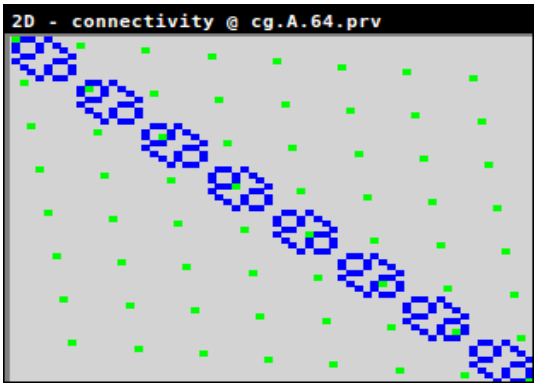

FIGURE 3.2: *Paraver view of p2p communication of cg benchmark with 64 tasks*

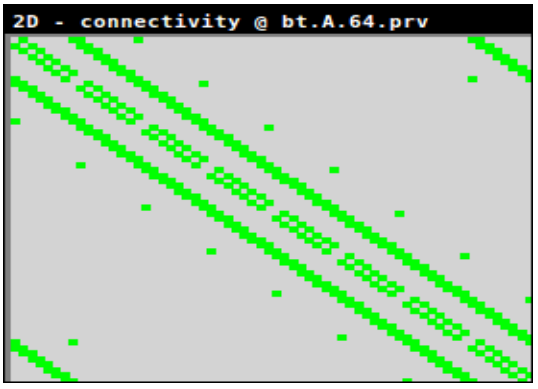

FIGURE 3.3: *Paraver view of p2p communication of bt benchmark with 64 tasks*

In the figures 3.1,3.2 and 3.3 the two colors "blue" and "green" have two different meanings, Blue color means more number of communication which are heavy and takes lots of time and the green ones indicate the light one which takes less time. The X and Y axis in those figures indicates the task number. In fig 3.1 one blue line is near to the center and another one is far from the center, the near one means short distance communication and vice versa.

The table 3.5 shows the applications that we have used and their sizes.

All three benchmarks are generated in the Marenostrum-IV with help of Extrae. The generated trace file was simulated with Dimemas in different environment and system configuration.

Name	Size	tasks per node
MG	64,128,256 and 512	16
CG	64 and 128	16
BT	64	16

TABLE 3.5: *benchmark used and with it's size*

3.5 Experiments

By using the above configuration we have done experiments with the NAS applications. We have developed a automated system, which is able to run simulation automatically as many times as we want. It is also able to generate the random files with given inputs, and check for the different system configurations and also checks for the trace files of the applications. The developed system not only run the application it also collect the informations and data's that we wanted to analyze and saved in compatible format. The automated system is based on bash and python programming language. The "Experiment" algorithm is based on bash and responsible to run the simulation and collect the data and also save those data in compatible format, where the "Random script" is based on python script which take care to write random files according to the input given by user.

Algorithm 4 Experiments based on bash script

```
1: for applications do
2:     for configurations do                          ▷ dimemas configuration file
3:         run Random script to generate random files
4:         for every random file do
5:             run the simulation 1..N
6:             collect the data with running the paramedir and save it in compatible format
7:         done
8:     done
9: done
10: End Experiment
```

The algorithm 4 shows step by step exactly how every experiment is executing. The experiment script firstly check for the applications, then for the configuration files that are for dimemas simulation and run the random script with given parameters by user. When the random files are created, then for each random files it run the simulation. Right after the simulation ends the paramerdir is run to collect the data of that simulated applications. For every random files the simulation can be repeated as we want. In

our case we have repeated 50 times the simulation for each random files. So for 6 different configuration we have generated 12 different random files and each random files are simulated for 50 times. 6X12X50 = 3600 is the number that each application is simulated.

To run the "Random script " which is explained by algorithm 5, we need to give three different values the start the end and the counter by which we want to increment value for every new random file eg: *"random script" <from ><to ><with >*. As in algorithm it takes a input values and set a ranges from where and up to where we want to extend our files(values inside a file) and with what increment for each file. At first it will check either the from values is less then to or not, if condition fails it will stop and exit the generation process. If the condition agreed then it check in which parameter it have to change the value, change it and increment "from" with "step" and repeat the step again till the condition fails.

Algorithm 5 Random script

```
1: Random script <from ><to ><step >
2: set_values = range(from, to, step)
3: for <from to to> do
4:     if "from" >= "to" then
5:         stop and exit
6:     else
7:         write the from value as parameters in indicated place
8:         write a file and end
9:     from ← from+step
10: done
11: End
```

3.6 Analyzed Parameters

We have collected a huge amount of data from our simulations with the help of paramedir and our automated system. Among all those data's we have chosen some fundamental factors, which are directly related to our interest and goal. The fundamentals parameters that we have collected and calculated to achieve our goal are explained below. The visualization tool "Paraver" is used to analyze and validate all these parameters.

1. **Communication Time**

This is the time taken by job to sending the message or receiving the message. In dimemas this can be obtained with equation 3.1.

$$CommunicationTime(C.T) = \frac{MessageSize(MS)}{AvailabeBandwidth(AB)} + Latency \tag{3.1}$$

As we are making penalties on the bandwidth this is the most relevant parameter to calculate. The Communication time is calculated by computing the ratio between size of the message to send with the available bandwidth and adding the latency in that. For every communication we are making a penalties in both bandwidth (inter and intra node), so the available bandwidth will be vary for each communication which will affect the C.T directly, which we want to see how it will change when we increasing the penalties.

2. **Computation Time**

   The time taken by the job which is not outside the communication is computation time. This factor is interesting when we are injecting the noise in the computation. While we are injecting the noise in the computation we are adding some amount of time in this time duration.

3. **Execution Time**

   The execution time is the summation of communication and the computation time. This factor will be affected in any case of noise injection and will give the overall preview of the application performance.

4. **Parallel efficiency(μ)**

   This factor shows the parallel efficiency of the application. It is literally product of load balance and communication efficiency and can be obtained as follows:

$$\mu_{||} = LB \times CommEff \tag{3.2}$$

   Parallel efficiency is measured in % which ranges 0 to 100, if obtained value is near to 100, it means there is very good parallelization and vice versa. The Communication Efficiency calculates the weight of MPI time due to actual communicatioin while the LB evaluates MPI time due to load balance. Combining these two factors we can get the parallelization of any application.

5. **Communication efficiency**

   Another important factor that we are analyzing is the communication efficiency. This can be calculated by computing the ratio between maximum computing time

taken from all the processors ($Max(t_i)$) over total time span of the computation(T).

$$CommEff = \frac{Max(t_i)}{T} \tag{3.3}$$

In HPC where the applications are executed in parallel, communication between cores and nodes is the factor which effects the scalability of any applications. Paraver normally shows this factor in %, the value nearer to 100 means, during the execution the communication efficiency of application is really good. In our result we are also calculating and showing this parameter in % like in paraver.

6. **Load Balance(LB)**

   Load balance is mostly affected due to the noise in the computation, because as we increase the execution time of one processor, the job will remain more time in the processor which is directly proportional to the time. It is also affected by the number of processors.

   It is defined as the ratio between the sum of computing time by all the processes ($\sum_i t_i$) by number of processors (P) and maximum time taken from all the processor ($Max(t_i)$). Which reflects the potential efficiency loss caused by imbalance in the total computation time by each processes. This factor is affected usually in parallel execution where the processes have to wait for the slowest one to catch up.

$$LB = \frac{\sum_i t_i}{P \times Max(t_i)} \tag{3.4}$$

   As it shows the difference in computing time of each thread. If the value is nearer to 1, it means non of the task have to wait on the synchronization. If the value is far from 1 or near to 0 then there is high variation and the tasks have to wait during the synchronization.

We have fetched out all these parameters to see the effect of noise in dimemas simulation. Paramedir is used to fetch out all these parameters and the applications from NAS benchmarks are being analyzed. After collected and analyzed there parameters we have generated some results, which is explained in the chapter 4.

Chapter 4

Experimental Results

To achieve our goal we have did our of experiments for 3 different applications MG,CG and BT from NAS. with 6 different configuration and 12 different random files to make penalties in communications and 10 random files to inject the noise in the computation, all the simulation has been repeated for 50 times. Each application has been simulated for four different configuration eg: MG(Multi Grid) application, which have 64 number of tasks in total were executed with penalties in inter-node communication intra-node communication, on both(intra and inter node) communication and finally in the computation. The same MG application with 128 number of tasks is also executed with the same procedure. For rest of the application also we have repeated the same process. The results after executing all those experiments are explained in this chapter.

The color identification for mapping tasks per nodes can be seen in the top right corner of the each figure. The text color which indicates the mapping of tasks per node represents vertical lines for that line. for example in fig 4.1 the green lines represent the 1 tasks per node and the text color of "1ppn"[1] belongs to the green line.

We have started making penalties on the network communication ie: in inter node communications. To do so we have started with the small size trace file of MG with only 64 tasks in total. The first simulation was without noise, which was executed several times to check the simulation is giving us a same result without any penalties or adding any noise. After that every time we have increased the deviation with fix mean of 300 as mentioned in previous chapter. X-axis represent the variation on deviation that we have increased every time with fixed mean.The values in X-axis are MB/s. The Y-axis represents efficiency in %. The Y-axis will be in time when the graph will be of

[1]Process Per Node

communication time or computation time or execution time. For every value we have executed the simulation 50 times, to see the variation in random value generation.

4.1 MG

Starting from the MG(Multi Grid) application we made a penalties in memory, network and in both communication which has 128 tasks in total. The figure 4.1 shows the experiment which was done with 16 tasks per node which means 128 tasks are mapped in 8 different nodes. With this experiment we expected to see the general behavior of mg application, because the tasks are distributed as like in original trace file. The figure 4.1 shows the distribution of execution time where x-axis shows the penalties in MB/s and the Y-axis shows the time in microseconds. Green and Pink line belongs to memory and network penalties respectively which shows almost same behavior. Where blue line shows the effect when we put penalties on both the communication which start to suffer earlier and also much more then other two. With this result we can say the MG application has long and short distance communication almost equally distributed.

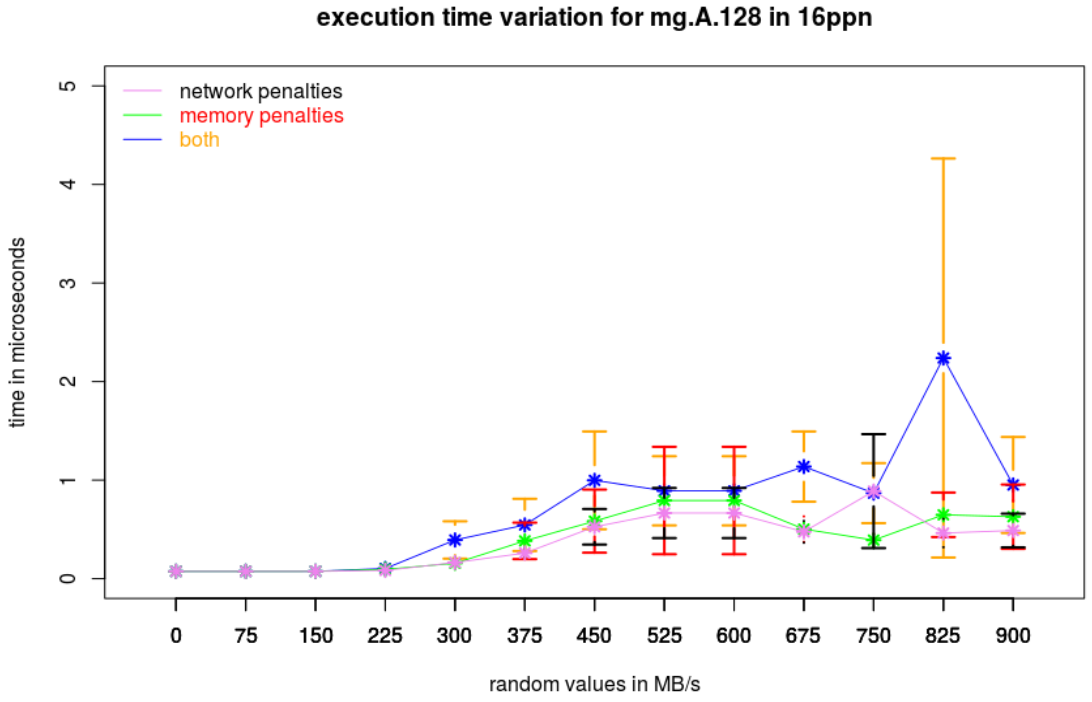

FIGURE 4.1: *Communication time variation of mg.A.128 trace file while simulating 16 task per node*

4.1.1 Noise on Network Communications

The penalties in inter node communication seems be problem in application when the tasks are mapped 1 per node. The green line in graph 4.2 shows how the application suffers as the penalties increase in inter node communication. The red vertical lines over green line shows the range of maximum and minimum value at that point and the line is drawn in the mean point of each them.

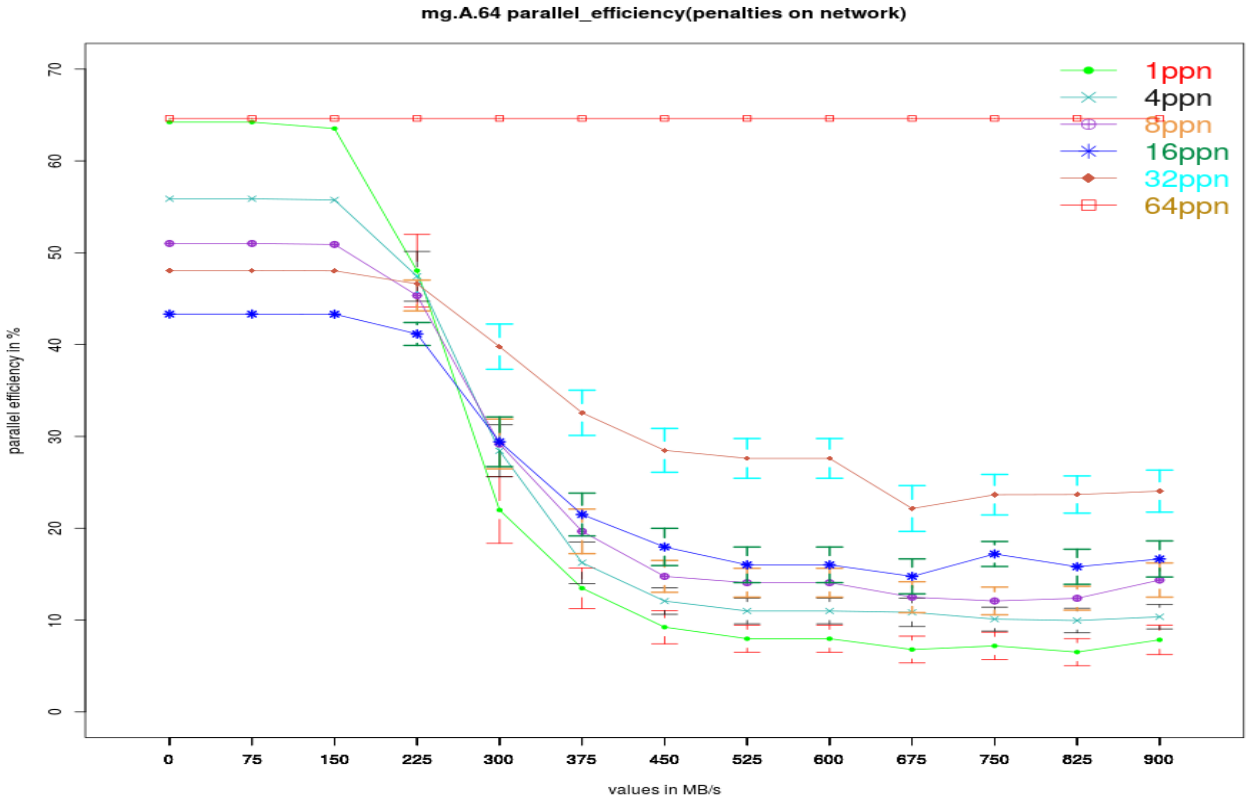

FIGURE 4.2: *making penalties in inter node communication in application mg with 64 tasks in total*

The red line which is steady even when the penalty rise till maximum is because this line shows when all the tasks are mapping in the same node and penalties is making in inter node communication, so the effect is null. On other hand the affect of penalties can be seen from third point in green one. The green line shows the affect of penalties in bandwidth, when we are mapping one task per node. In this scenario we are making penalties on inter node communication so this configuration is the one which suffered the most.

The interesting part of the graph 4.2 is, the green and red lines started from the same point and goes together till second point where they are almost equal, where the variability of random value is 75 MB/s. From third point the green line started suffering

little bit and at fourth point it suffered drastically and it never stop to falling down. When we are simulating with configuration 32 tasks per node which means only two nodes are communicating, The parallel efficiency does not seems that much affected comparing to others. The dark blue line which starts from the button ends up in the third position at the end "counting from top" which is when the configuration is 16 tasks per node. This configuration also seems good option for this kind of application when we have lots of noise in network. All the lines started to decline from third point which has mean of 300 MB/s and variation of 150 MB/s. From this we can conclude that this application is not using very high bandwidth.

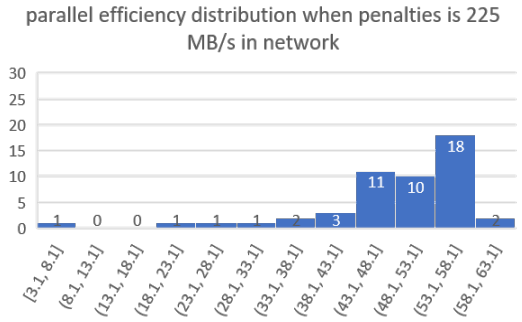

FIGURE 4.3: *distribution of parallel efficiency at point 225 of green line of figure 4.1*

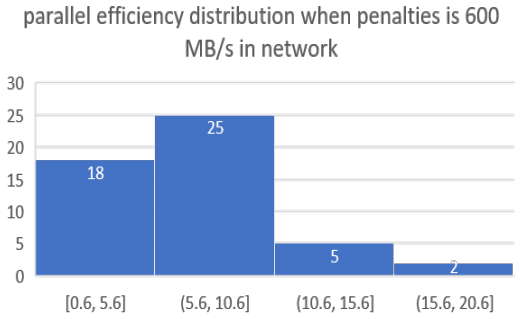

FIGURE 4.4: *distribution of parallel efficiency at point 600 of green line of figure 4.1*

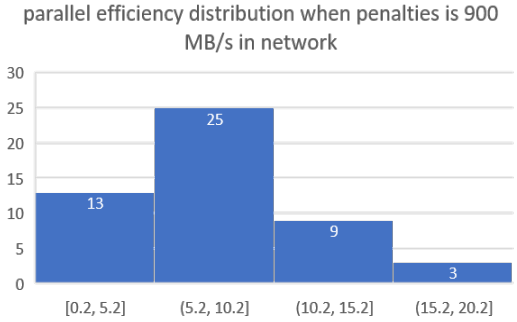

FIGURE 4.5: *distribution of parallel efficiency at point 900 of green line of figure 4.1*

figure 4.3,4.4 and 4.5 shows the distribution of parallel efficiency when the penalties in 225,600 and 900 MB/s. The Y-axis shows number of occurrence and X-axis represent the group of parallel efficiency with difference of 5. We have repeated 50 times the simulation for one input value for every point. With these bar graphs we can see how the parallel efficiency are distributed even we are giving the same input. This show the variation of parallel efficiency was due to little and frequent noise then the large and constant noise.

In graph 4.6 where we have simulated the MG with 128 tasks in total,here we are analyzing the communication efficiency. The green line behaves same as in graph 4.2

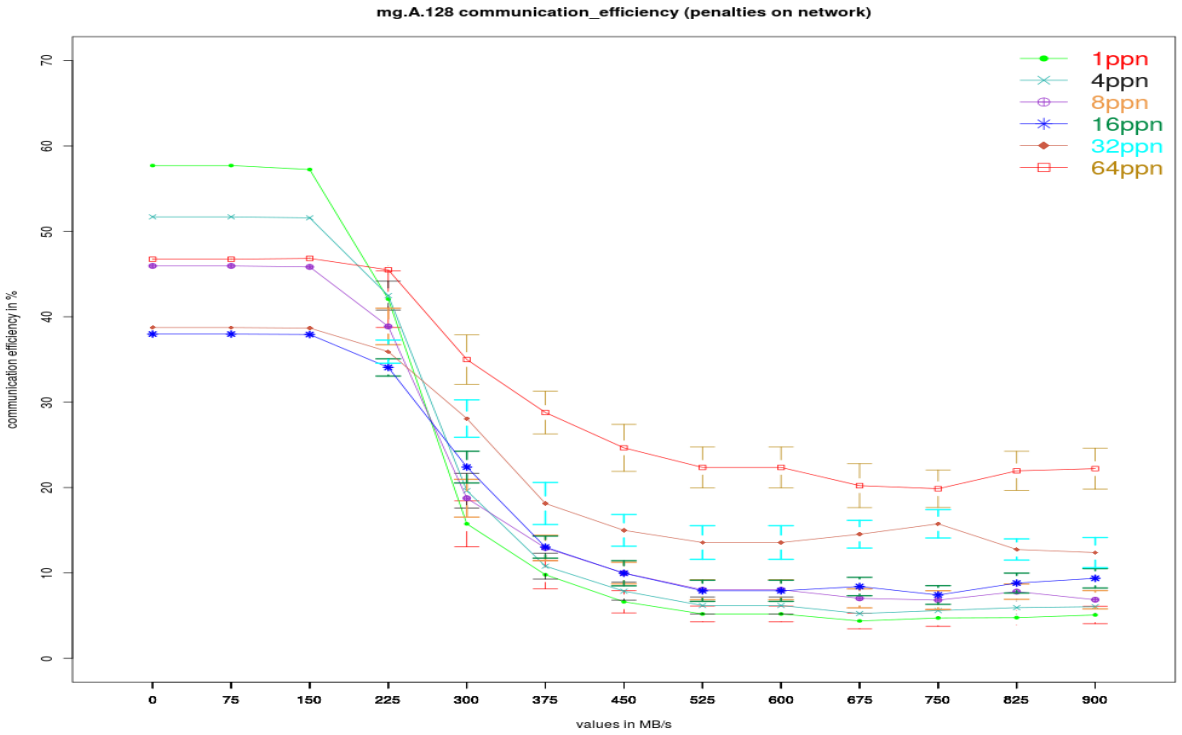

FIGURE 4.6: *communication efficiency while making penalties in inter node communication in application mg with 128 tasks in total*

but the red lines start to scale down from fifth point which is when penalties in 300 MB/s. As we have 128 tasks in total and red lines represent the configuration of 64 tasks per node. Only two nodes are involved in inter node communication, so this configuration is less affected one when we put penalties on network.

4.1.2 Noise in Memory Communications

Figure 4.7 and 4.8 both shows communication efficiency when we make a penalties on memory communication with 128 and 256 tasks in total of MG application. In both case green line is unaffected where the red one is most affected as expected. In figure 4.8 the lines at the end are much more compacted comparing with the 4.7 from which we can conclude that the bigger application suffers much more in any configuration. If we concentrate on the dark blue line in figure 4.7, with 0 noise the communication efficiency is lowest one but when we are injecting the maximum noise the communication efficiency is almost equal to the 8 tasks per node which is indicating by violet color line. Almost the same case can be seen in graph 4.8, the violet line starts from the higher communication efficiency and ends up with same communication efficiency as the dark blue one. With this two graphs we can say that 16 tasks per node configuration is good option when there is noise in memory communication for medium size applications.

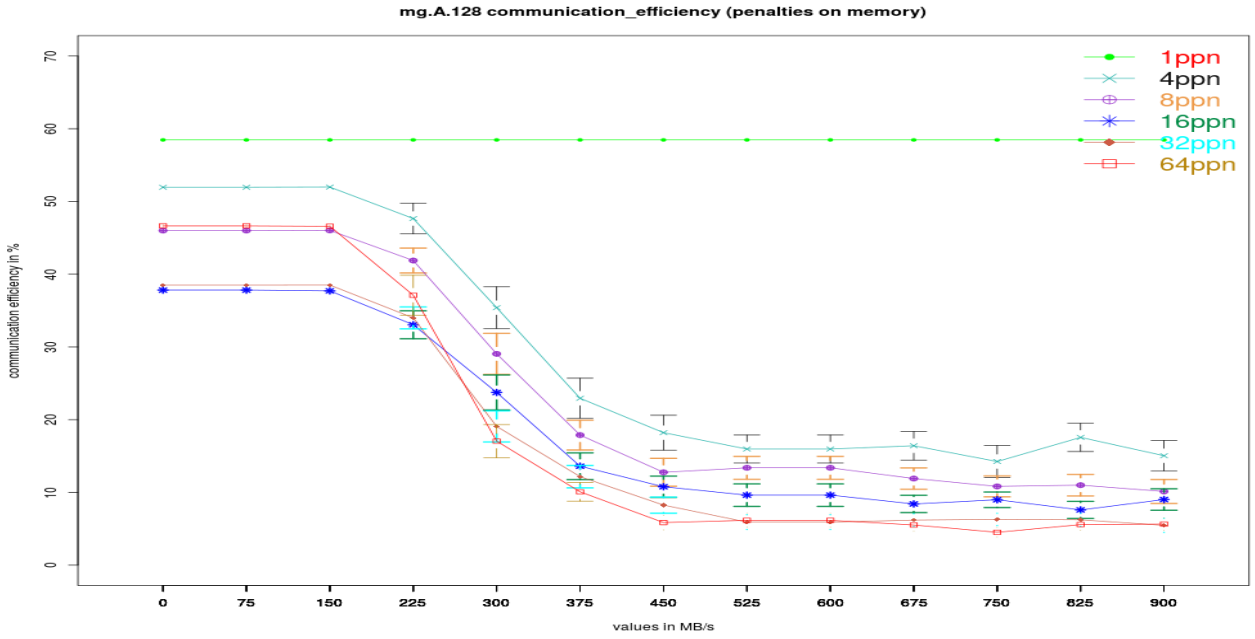

FIGURE 4.7: *communication efficiency while making penalties in intra node communication in application mg with 128 tasks in total*

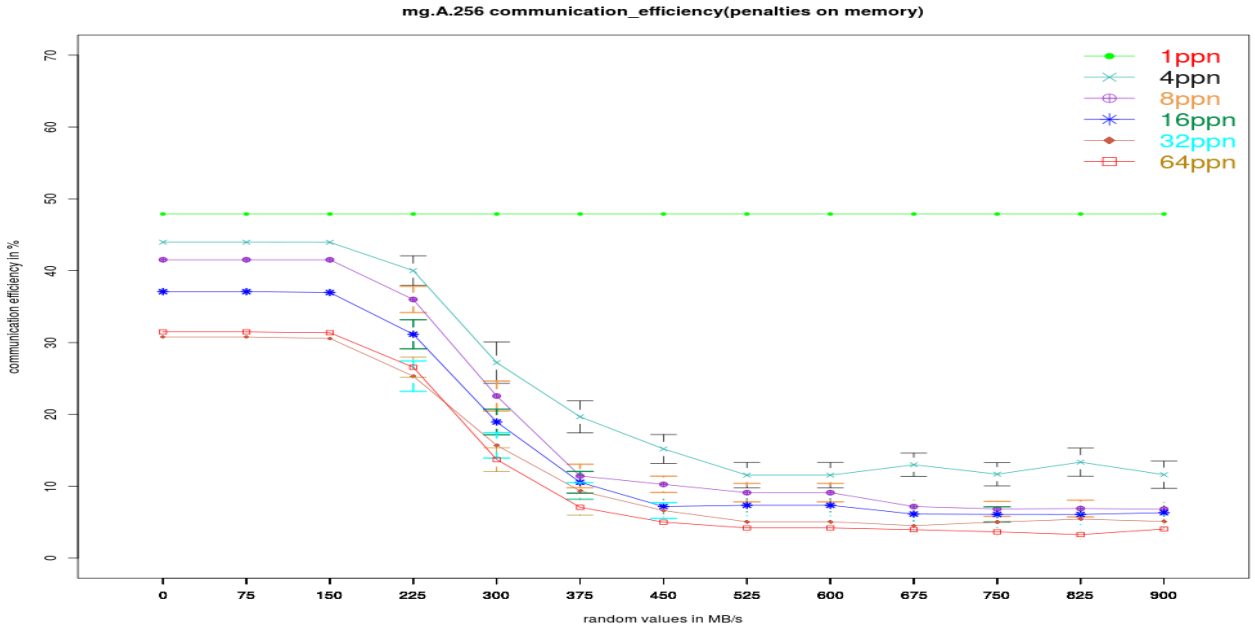

FIGURE 4.8: *communication efficiency while making penalties in intra node communication in application mg with 256 tasks in total*

4.1.3 Noise in Both(network/memory) Communications

Figure 4.9 and 4.10 are when we make penalties on both communication with 64 and 512 tasks in total respectively. In figure 4.10 the red line has lowest parallel efficiency and green has the highest at beginning but as the penalties get increasing both of them

start to suffer. The gap between red and green line, is higher when penalties in 150 MB/s then 225 MB/s because the green line has declined much more when the penalties in 225 MB/s which shows it suffers much more. Now we can say that Mg is equally sensible towards noise in memory and in network.

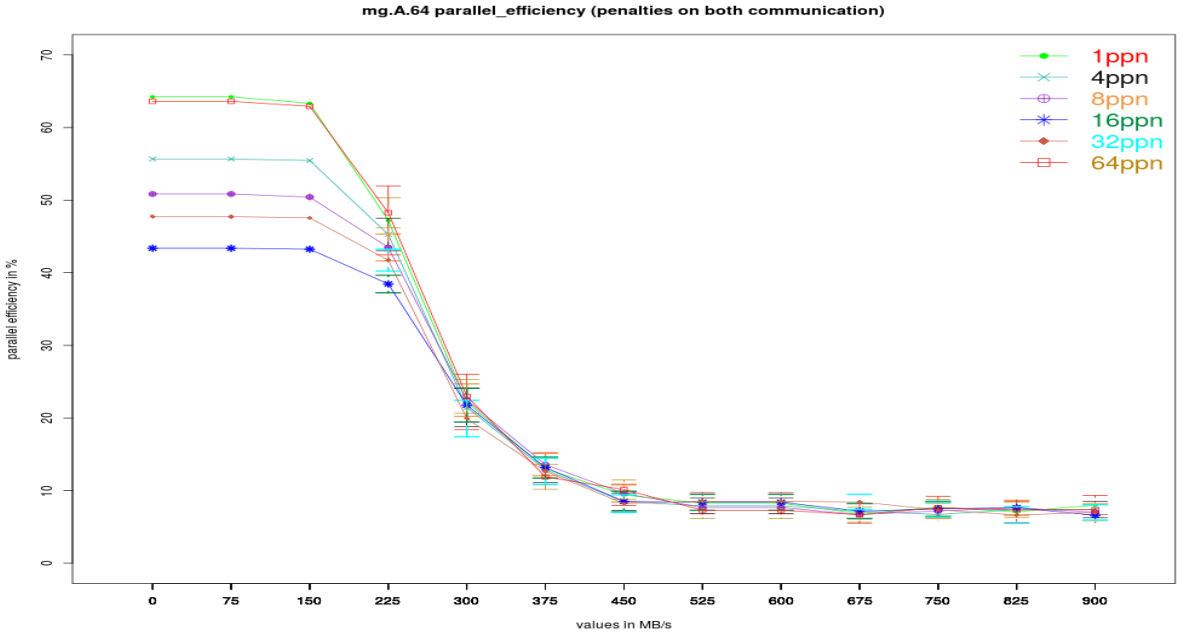

FIGURE 4.9: *parallel efficiency while making penalties in both communication in application mg with 64 tasks in total*

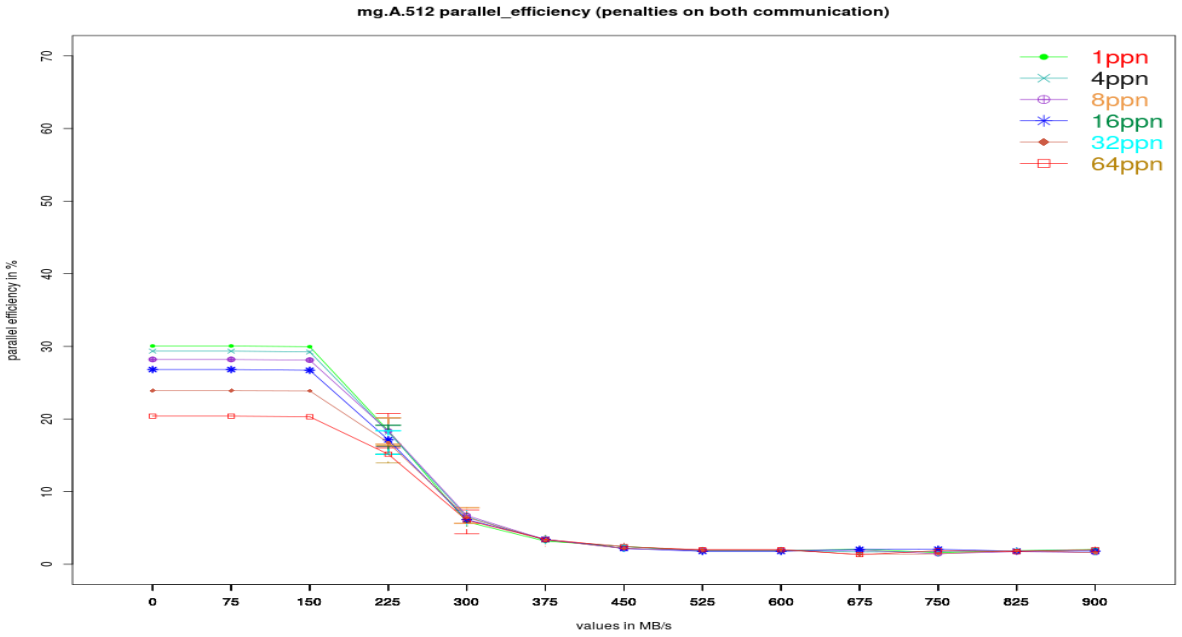

FIGURE 4.10: *parallel efficiency while making penalties in both communication in application mg with 512 tasks in total*

4.1.4 Noise in Computation

In figure 4.11 x-axis show the values in nanoseconds and the Y-axis show the time in micro seconds. The configuration is almost independent towards the noise in the computation even though we can see little bit variation at some points. Even the little variation of the execution time variation the parallel efficiency is vary with notable range. More factors are needed to be discovered explain this kind of phenomena.

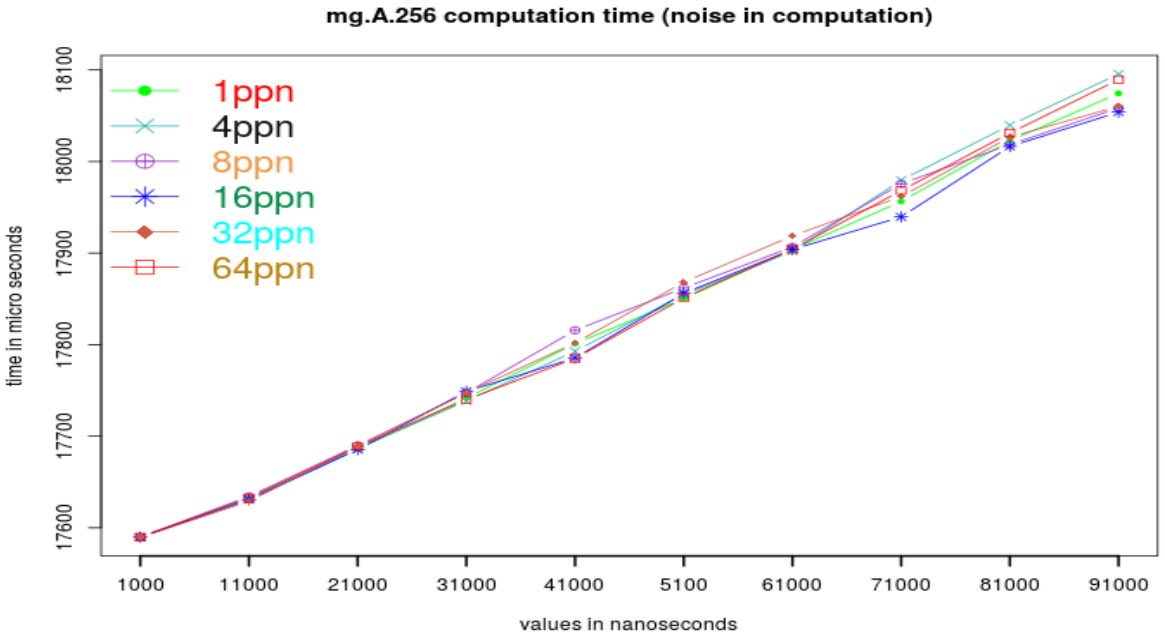

FIGURE 4.11: *computation time variation for different configuration while injecting noise in the computation*

4.2 CG

The figure 4.12 is the execution time for CG application which have 128 number of tasks in total, we have executed it with the configuration of 16 tasks per node, making penalties on network,memory and both communication. The blue line represent penalties on both communication where the green line represent on memory and the pink line represent when the penalties are in network. Using 16 tasks per node configuration we are using 8 nodes in total, which means communication between inter and intra nodes are relatively equal. In figure 4.1 we can see the variation of execution time in MG application with same configuration where both the communications are equally distributed. In figure 4.11 with same configuration while making the same penalties the memory communication seems affected much more then the network. The green lines is very close to the blue line, from this we can conclude that this application has more short distance communications than long.

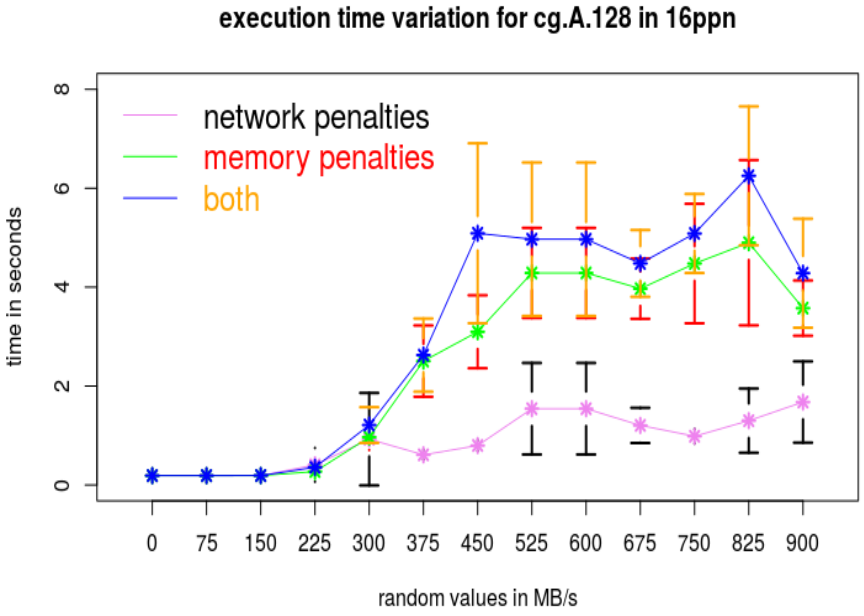

FIGURE 4.12: *Execution time variation when incrementing noise in cg with 128 number of tasks in total simulating with configuration 16 tasks per node*

The red vertical line shows the variation when we execute the application with same amount of penalties in memory communication. The variation is much more higher than the black vertical lines which represent when we put the penalties in network communication. Similarly the orange lines over dark blues line shows the variation when the penalties are in both communication, which vary even higher then red lines. This also explains the sensibility of applications towards communications.

4.2.1 Noise in Network Communication

The penalties on network is always costly when the configuration is 1 task per node, figure 4.13 also explains the same thing. The green line start from the top and ends at the bottom. If we focus on the dark blue line which starts from the fourth position but ends up in the third. This configuration might be the better option for application with this size and characteristics.

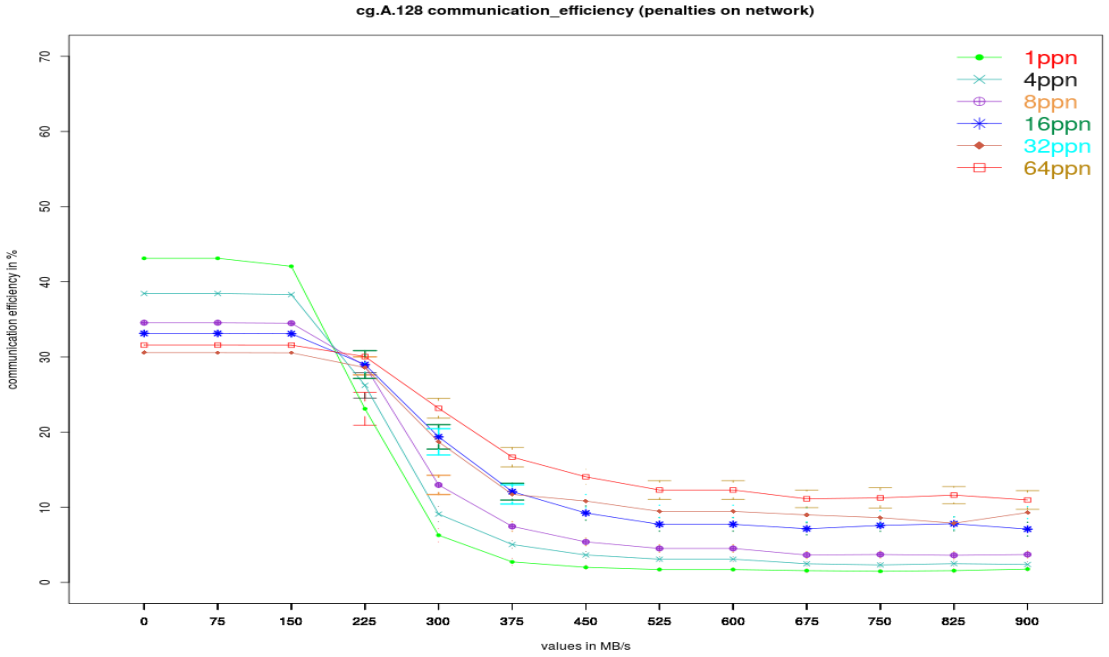

FIGURE 4.13: *parallel efficiency while making penalties in network communication in application cg with 128 tasks in total*

4.2.2 Noise in Memory Communication

In figure 4.14 green line remains unchanged but the red line start declining from the third point when there is penalties of 150 MB/s, this also proves that this application has more near communications. As we put more than one task per node the parallel efficiency start to decrease rapidly.

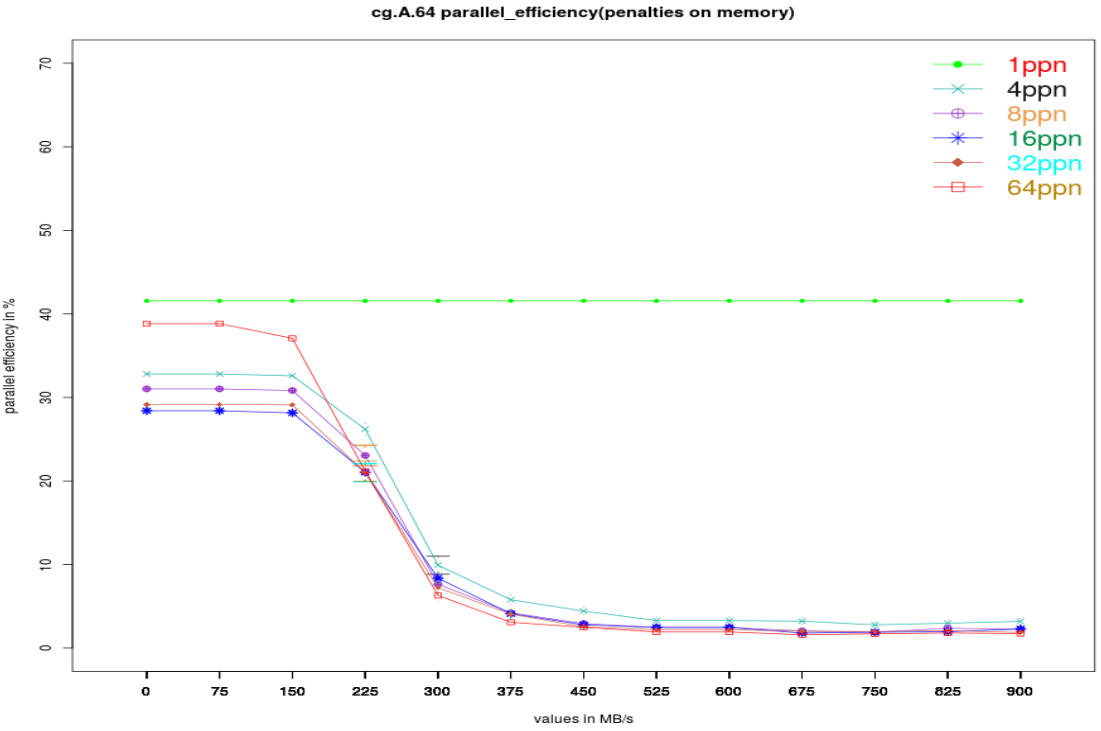

FIGURE 4.14: *parallel efficiency while making penalties in network communication in application cg with 128 tasks in total*

4.2.3 Noise in Both(network/memory) Communication

As we put the noise in both communication, which is shown in figure 4.15 the red and green lines behave almost similarly. Both of them start to come down from the same point, and ends up at the same point, so in case of existence of noise in both the communication, with this kind of configuration we cannot get better performance. Another interesting case in the same figure is the dark blue and the orange line, at point 225 the orange line falls down crossing the dark blue line. From this two line we can say that in case of noise in both communication the configuration really does matter in communication and parallel efficiency of the applications. Figure 4.16 and 4.17 are to show the distribution of communication efficiency when we simulate the CG application with dark blue and orange line configuration from figure 4.15 with penalties of 225 MB/s.

Which also shows that with 16 tasks per node configuration we can obtain the better performance in case of noise in both the communications.

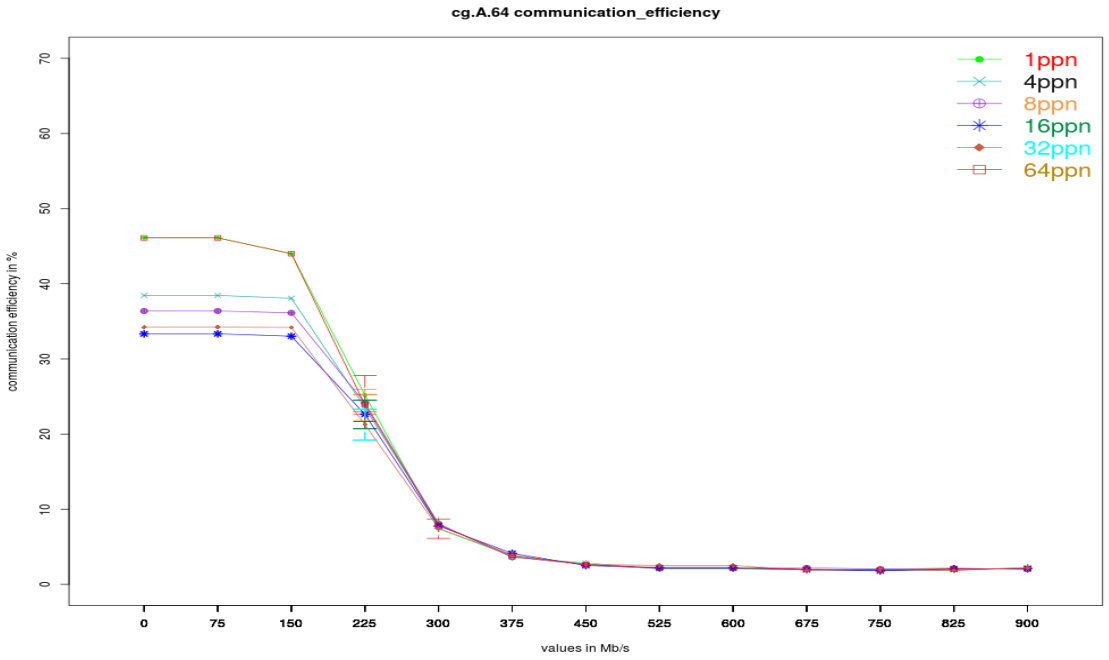

FIGURE 4.15: *parallel efficiency while making penalties in both communication in application cg with 64 tasks in total*

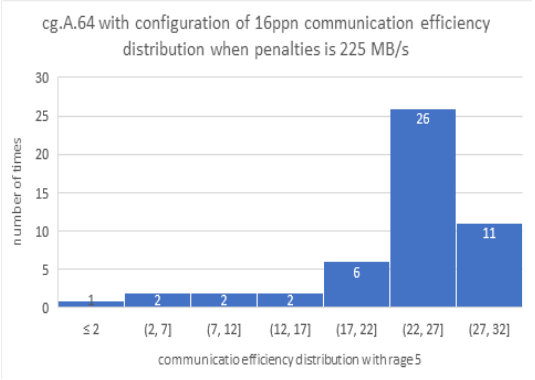

FIGURE 4.16: *distribution of communication efficiency of blue line configuration when penalties is 225 MB/s*

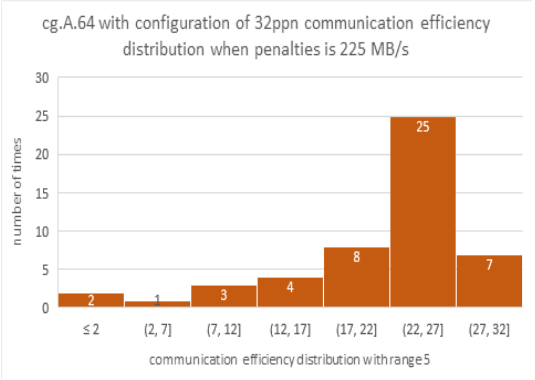

FIGURE 4.17: *distribution of communication efficiency of orange line configuration when penalties is 225 MB/s*

4.2.4 Noise in Computation

Figure 4.18 shows the parallel efficiency of CG application which have 128 tasks in total when the noise is injecting on the computation. when injecting noise in the computation the load balance varies which is also one of the reason that the parallel efficiency affected beside communication.

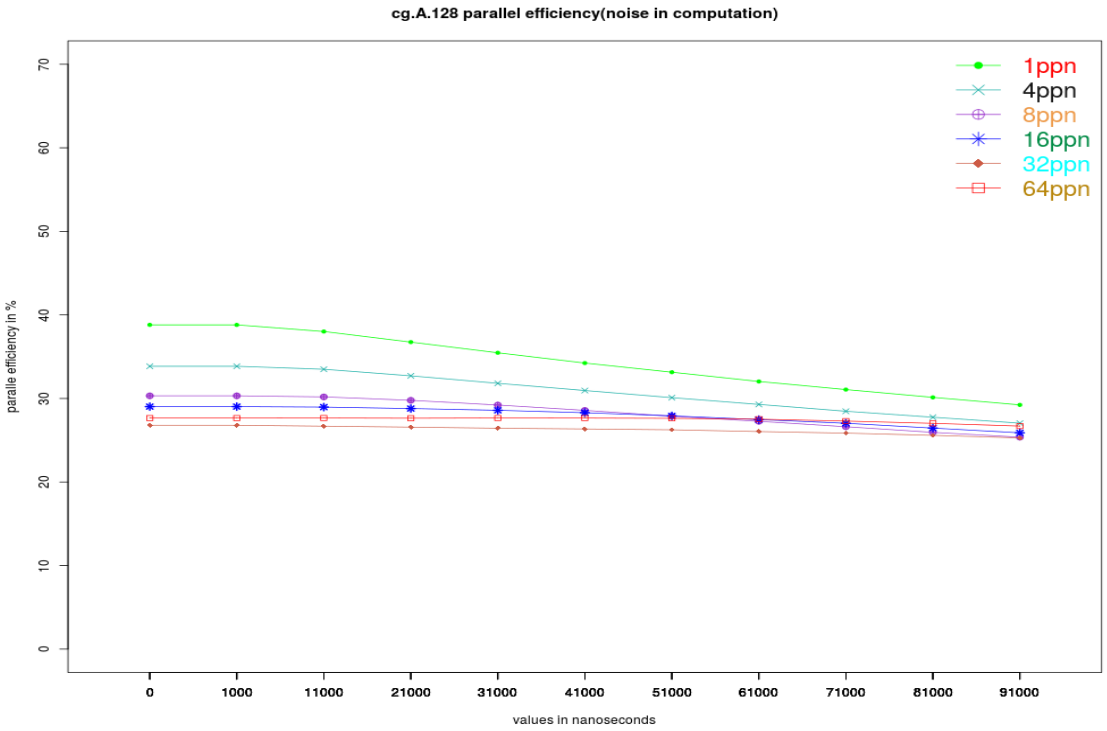

FIGURE 4.18: *parallel efficiency variation when injecting noise in the computation of CG application with 128 tasks in total*

4.3 BT

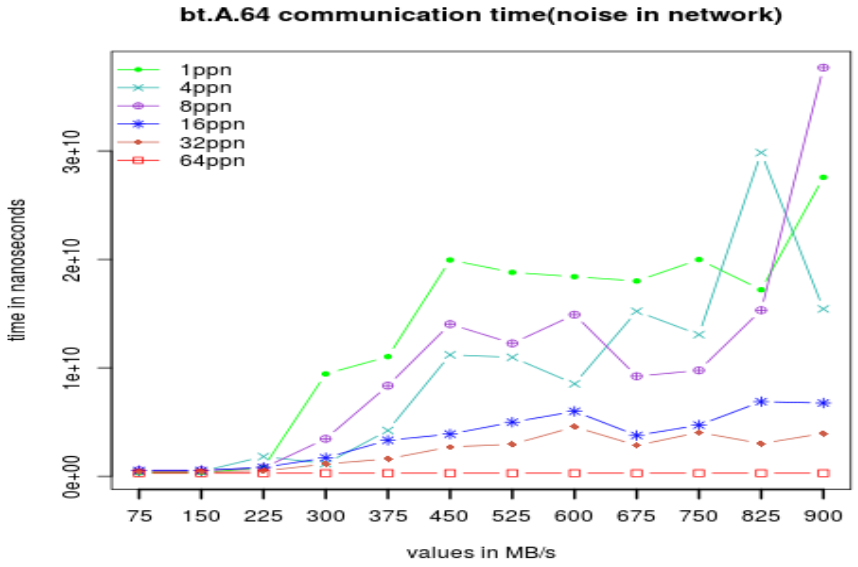

FIGURE 4.19: *Communication time while making penalties on the network in BT application of 64 task in total*

4.3.1 Noise in Network Communication

The figure 4.19 shows the communication time for BT application when we are making penalties on the network. The green line is the most affected one as in other applications. Here the interesting part is the light blue line and the violent line, which rises almost equally but at the end the violet goes up drastically. This affect is because the application is bearing both contention and the penalties in communication. As the contention is 1 for all configuration in the violet configuration has 8 tasks per node which are mapped in 8 different nodes, if 8 tasks from one node wants to communicate to the remaining tasks in all nodes then there is high probability that they will suffer from both contention and the penalties on the network. This kind of configuration gives us the worst efficiency if we have noise and the contention.

4.3.2 Noise in Memory Communication

Making penalties on the memory affects when we are putting more than one tasks in a node. Figure 4.20 shows when we make a penalties on the memory communication in BT applications of 64 tasks in total. Performance of red line goes down as the penalties goes up as expected, but the behavior of dark blue, orange and the violet lines are quite interesting in this figure. The dark blue and violet lines start from the same point, but the orange line start from above. As the penalties reach at 225 MB/s the orange line shrinks down much more comparing to other two lines. If we go one step ahead when the penalties is 300 MB/s the gap between three line is huge comparing the previous point. The orange line goes down as much as the red as well the dark blue line but the violet line maintained the performance. Which shows that the parallel performance of application not only depends on the amount of noise but also in the configuration.

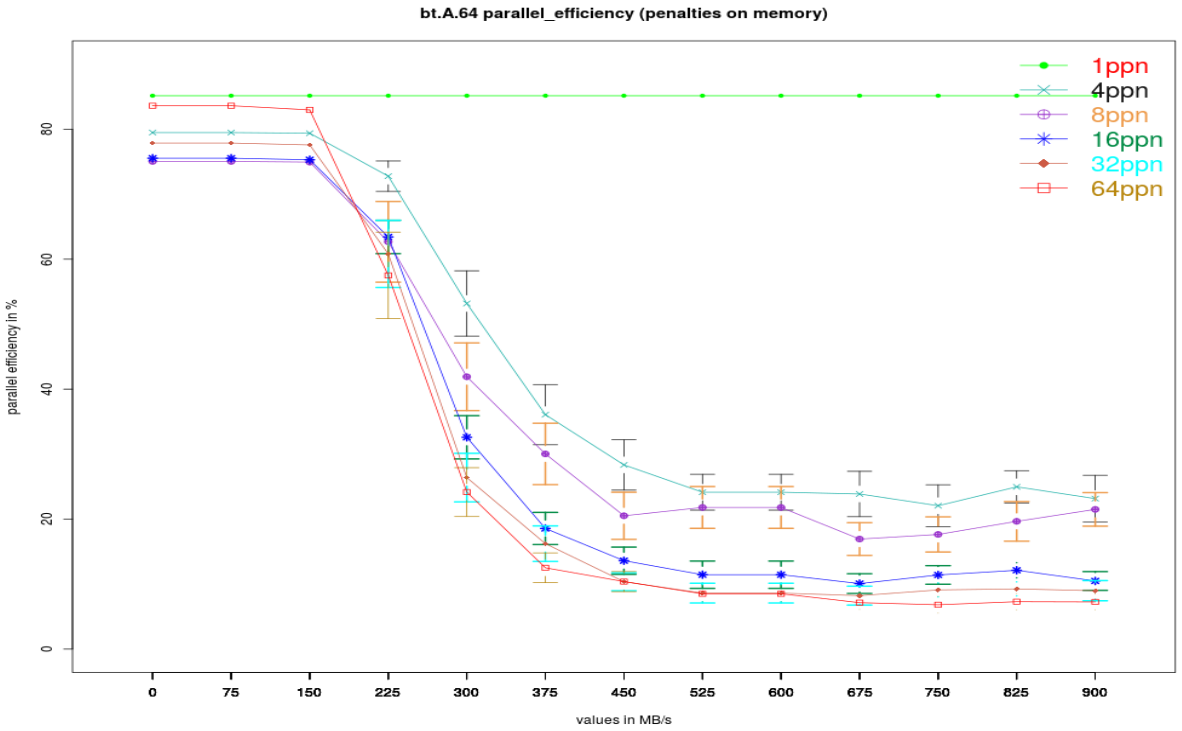

FIGURE 4.20: *parallel efficiency while making penalties on the memory in BT application of 64 task in total*

4.3.3 Noise in Both(network/memory) Communication

In figure 4.21 we can see how sensible the BT application towards noise in the both communications. The red line shows the the application is using less bandwidth for short distance communication while the green and light blue line explains use of high bandwidth for a long distance communications. The dark blue and violet lines are using both communications (network and memory) so they are suffering much more with the penalties on communications.

With these two figure 4.22 and 4.23 we can see how the communication time is distributed when the penalties is maximum in both the communications. In both of these figures X-axis is time in seconds grouping with 10 seconds and the Y-axis is the occurrence of that value. If we see in the figure 4.23 the execution time are much more distributed and larger than figure 4.22. The blue line configuration is the worst case scenario for this application when the noise is in both communications.

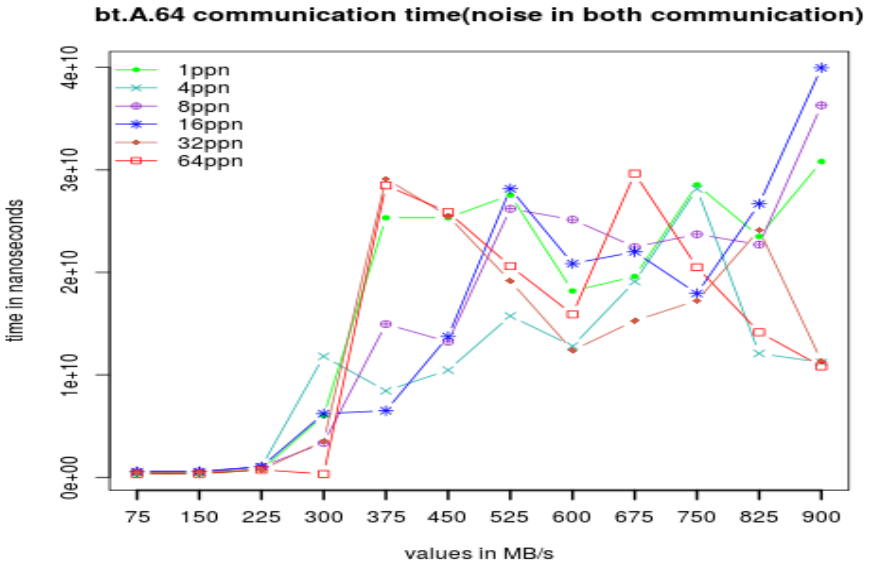

FIGURE 4.21: *Communication time while making penalties on the both communication in BT application of 64 task in total*

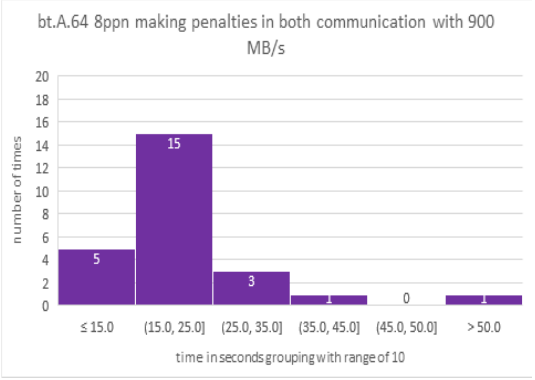

FIGURE 4.22: *distribution of communication time of violet line at point where communication penalties in 900 MB/s*

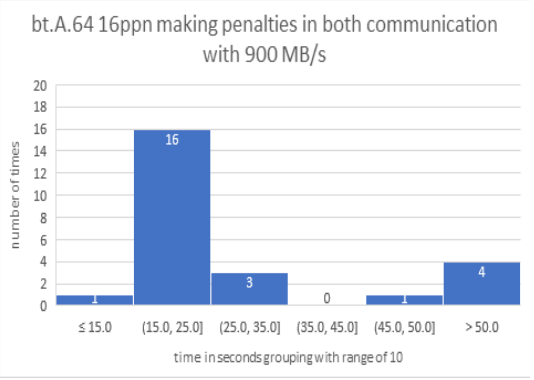

FIGURE 4.23: *distribution of communication time of blue line at point where communication penalties in 900 MB/s*

Chapter 5

Conclusion

Many studies has been done to observe the performance of MPI applications on HPC's but this research is focused to analyse the sensibility of parallel applications towards noise. We have added a noise model in Dimemas MPI aplications simulator, from which we were able to see the effect of noise in the applications. The noise has been injected in the communication (network and memory) as well as in the computation as a Gaussian distribution with different sigma, to see how sensible are the application towards noise. Three applications (MG, CG and BT) with different characteristics have been selected from NAS benchmarks. The experiments have been done with 6 different mapping configurations. The MG application has long and short distance communication distributed equally so noise in any of the communication has almost equal effect and when the noise is injected in both network and memory communications, the parallel efficiency is reduced almost by half. In the case of CG the noise in the memory has a higher impact than then noise in the network. The BT application shows a similar behavior than MG.

Various factors such as communication time, execution time, parallel efficiency and communication efficiency are identified and used to measure the sensibility of the applications which is also the important aspect of this research project. The same model and the factors are used to inject and measure the noise in the computation, and same applications are simulated to analyze how sensible they are with noise in the computation. Another aspect that we have observed during the project was as the the size of the applications and the parallel efficiency are inversely proportional. That means as the size of the application increase the parallel efficiency decrease.

The major factors which effect in the performance of the MPI applications has be identified and studied. A successful methodology is implemented to collect all those

factors automatically, which is also the strong part of this research. Three application are analyzed using this model to validate that the model is implemented correctly. As the model works properly, other applications could be analyzed including huge size benchmarks. which was left as a future work.

Bibliography

[1] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. A quantitative analysis of os noise. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 852–863, May 2011.

[2] TOP500.org. *top500 supercomputer in the work*. 2017.

[3] cray-1 research inc. *The cray-1 computer system*. 1977.

[4] DOMENICO TALIA. Models and trends in parallel programming. *Parallel Algorithms and Applications*, 16(2):145–180, 2001.

[5] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[6] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 55–, New York, NY, USA, 2003. ACM.

[7] Marc Casas, Rosa Badia, and Jesús Labarta. Automatic analysis of speedup of mpi applications. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 349–358, New York, NY, USA, 2008. ACM.

[8] ASC Sequoia Benchmark Codes. *top500 supercomputer in the work*. june 2013.

[9] Nawal Copty Marty Itzkowitz, Oleg Mazurov and Yuan Lin. Coarray c++.

[10] Troy A. Johnson. An openmp runtime api for profiling.

[11] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Jose Maria Cela. A parallel program development environment. 04 1995.

[12] E.; Barton J.; Browning D.; Carter R.; Dagum L.; Fatoohi R.; Fineberg S.; Frederickson P.; Weeratunga S. Baily, D.; Barscz. Nas technical report rnr-94-007,. pages 349–358. NAS, March 1994.

[13] J.E. Ayers. *Digital Integrated Circuits: Analysis and Design.* Taylor & Francis, 2003.

[14] Kaladhar Radhakrishnan Michael J. Hill Kemal Aygün Chia-Pin Chiu Gaurang Choksi. *Optimization of Package Power Delivery and Power Removal Solutions to meet Platform level Challenges.* intel developer forum, 2003.

[15] J. Choi, B. Beker, and C. Hilbert. Noise distribution of multi-core cpu packages. In *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems*, pages 209–212, Oct 2010.

[16] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. "system noise, os clock ticks, and fine-grained parallel applications". In *ICS*, 2005.

[17] Wolfgang Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. Vampir: Visualization and analysis of mpi resources. 12, 05 1996.

[18] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 371–377, Sept 2004.

[19] Mathieu Desnoyers and Michel R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. 2006.

[20] William T. C. Kramer and Clint Ryan. Performance variability of highly parallel architectures. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Yuriy E. Gorbachev, Jack J. Dongarra, and Albert Y. Zomaya, editors, *Computational Science — ICCS 2003*, pages 560–569, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[21] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12, Nov 2007.

[22] A. Nataraj, A. D. Malony, S. Shende, and A. Morris. Kernel-level measurement for integrated parallel performance views: the ktau project. In *2006 IEEE International Conference on Cluster Computing*, pages 1–12, Sept 2006.

[23] Department of Computer and LANL NM Research Centre Julich ZAM Germany Information Science, University of Oregon Advanced Computing Laboratory. *TAU (Tuning and Analysis Utilities)*. 2017.

[24] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki. Characterizing output bottlenecks in a supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[25] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 42:1–42:12, Piscataway, NJ, USA, 2008. IEEE Press.

[26] Van Emden Henson and Ulrike Meier Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155 – 177, 2002. Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss.

[27] Hormozd Gahvari, Allison H. Baker, Martin Schulz, Ulrike Meier Yang, Kirk E. Jordan, and William Gropp. Modeling the performance of an algebraic multigrid cycle on hpc platforms. In *ICS*, 2011.

[28] Claudia Rosas, Judit Giménez, and Jesús Labarta. Scalability prediction for fundamental performance factors. *Supercomput. Front. Innov.: Int. J.*, 1(2):4–19, July 2014.